CANTINA

Optimism: PolicyEngine-Staking Contract

Security Review

Cantina Managed review by:
Kaden, Lead Security Researcher
R0bert, Lead Security Researcher

March 11, 2026

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Optimism is a fast, stable, and scalable L2 blockchain built by Ethereum developers, for Ethereum developers. Built as a minimal extension to existing Ethereum software, Optimism's EVM-equivalent architecture scales your Ethereum apps without surprises. If it works on Ethereum, it works on Optimism at a fraction of the cost.

From Feb 24th to Feb 27th the Cantina team conducted a review of optimism on commit hash ea7595dd. The team identified a total of **10** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	0	0	0
Low Risk	1	1	0
Gas Optimizations	0	0	0
Informational	9	8	1
Total	**10**	**9**	**1**

2.1 Scope

The security review had the following components in scope for optimism on commit hash ea7595dd:

```
packages/contracts-bedrock
├── interfaces/periphery/staking/IPolicyEngineStaking.sol
├── scripts/deploy/DeployPolicyEngineStaking.s.sol
└── src/periphery/staking/PolicyEngineStaking.sol
```

3 Findings

3.1 Low Risk

3.1.1 Pause bypass through allowlist mutation

Severity: Low Risk

Context: PolicyEngineStaking.sol#L289-L296

Description: During a pause window, only staking and beneficiary changes are blocked while allowlist updates remain active. This allows a beneficiary to continue re-routing attributed stake by toggling `setAllowedStaker` / `setAllowedStakers`. That creates a state-change path during pause where ordering-power state can still mutate without new deposits/withdrawals.

Example: while paused, a beneficiary can still call `setAllowedStaker(staker, false)`. If that staker is currently delegated to the beneficiary, `_decreasePeData(msg.sender, ...)` and `_increasePeData(staker, ...)` execute immediately, moving effective stake attribution back to the staker and changing ordering-power state during pause.

Relevant snippets:

```
modifier whenNotPaused() {
    if (paused) revert PolicyEngineStaking_Paused();
    _;
}
```

```
function stake(uint128 _amount, address _beneficiary) external whenNotPaused { ... }

function changeBeneficiary(address _beneficiary) external whenNotPaused { ... }
```

```
function setAllowedStaker(address _staker, bool _allowed) public {
    if (_staker == msg.sender) revert PolicyEngineStaking_SelfAllowlist();

    allowlist[msg.sender][_staker] = _allowed;

    if (!_allowed) {
        StakedData storage stakedData = stakingData[_staker];
        if (stakedData.beneficiary == msg.sender) {
            _decreasePeData(msg.sender, stakedData.stakedAmount);
            emit BeneficiaryRemoved(_staker, msg.sender);

            stakedData.beneficiary = _staker;
            _increasePeData(_staker, stakedData.stakedAmount);
            emit BeneficiarySet(_staker, _staker);
        }
    }
}
```

This behavior conflicts with the design intent in:

- Design doc section Pause / Unpause: pause is meant to stop staking and beneficiary-changing actions while still allowing unstake.
- Design doc section DenyAllowed: disallowing a staker is documented as having no effect on existing stake or links.

Recommendation: Decide whether pause should be advisory or a full contract freeze and enforce it consistently. If full freeze is expected, add `whenNotPaused` to allowlist mutation entrypoints and any paths that can reattribute effective stake while paused. If not, document the nuance in contract comments, design docs and incident runbooks.

Optimism: Fixed in commit 3f85e88 by documenting that pause is advisory rather than a full freeze, so allowlist revocations remain permitted during pause and may reattribute ordering power back to the staker.

Cantina Managed: Fix verified.

3.2 Informational

3.2.1 `ChangeBeneficiary` reverts on same beneficiary despite no-op specification

Severity: Informational

Context: PolicyEngineStaking.sol#L239

Description: The implementation reverts when the requested beneficiary is already the current beneficiary:

```
if (currentBeneficiary == _beneficiary) revert PolicyEngineStaking_SameBeneficiary();
```

This behavior conflicts with the documented contract semantics in the design/spec flow, where `changeBeneficiary` is defined as an idempotent no-op when the beneficiary is unchanged. The spec/flow expects this path to return without state changes.

An external caller that retries or sends conservative/batch operations assuming idempotence can now get unexpected reverts and failed txs, even though no meaningful state mutation would be required.

References:

- Spec: `https://www.notion.so/defi-wonderland/Spec-2f09a4c092c781d89ba9f004f01a5cf5`.

Recommendation: Consider aligning with the documented no-op semantics by changing this branch to `return;`, or update the spec/design text to state explicit revert-on-no-op as an intended behavior.

Optimism: Fixed by updating the Spec documentation.

Cantina Managed: Fix verified.

3.2.2 Owner transferability conflicts with immutable-owner design intent

Severity: Informational

Context: PolicyEngineStaking.sol#L167-L173

Description: The contract exposes ownership transfer via the `transferOwnership` function. This is a behavior mismatch against the design/spec framing of an owner established at constructor time and treated as immutable.

Recommendation: Consider updating the design/spec to explicitly permit administrative owner migration and document required governance controls, or remove/disable ownership transfer if the intended model is immutable owner control.

Optimism: Fixed by updating the Spec documentation.

Cantina Managed: Fix verified.

3.2.3 Pause behavior mismatch for `changeBeneficiary`

Severity: Informational

Context: PolicyEngineStaking.sol#L229

Description: The contract gates `changeBeneficiary` with `whenNotPaused`:

```
function changeBeneficiary(address _beneficiary) external whenNotPaused { ... }
```

However, one of the formal spec documents describes `changeBeneficiary` as a non-paused operation ("Not subject to the pause modifier"), while other design references treat beneficiary changes as part of pausing controls. This creates ambiguity over expected operational behavior during incident pauses.

Recommendation: Consider choosing one intended mode and make it consistent across design/spec/FMA and implementation:

- If beneficiary changes must remain blocked during pause, update the spec wording to remove the no-pause exception.
- If they should remain callable while paused, remove `whenNotPaused` from `changeBeneficiary` and align docs and incident runbooks accordingly.

Optimism: Fixed by updating the Spec documentation.

Cantina Managed: Fix verified.

3.2.4 ABI/signature mismatch on stake/unstake and `stakedAmount` width

Severity: Informational

Context: PolicyEngineStaking.sol#L193

Description: The written spec still declares `uint256` for both primary state-transition amounts, but the implementation exposes `uint128`:

```
function stake(uint256 _amount, address _beneficiary) external
```

```
function unstake(uint256 _amount) external
```

```
returns (
    uint256 stakedAmount,
    address linkedTo
)
```

while the contract entrypoints are:

```
function stake(uint128 _amount, address _beneficiary) external whenNotPaused
```

```
function unstake(uint128 _amount) external
```

```
function stakingData(address _account) external view returns (uint128 stakedAmount_,
↪  address beneficiary_);
```

This is just a documentation/integration mismatch (and ABI expectation mismatch if the spec is treated as source of truth for integrations).

Recommendation: Consider aligning the spec and interface description to `uint128` end-to-end.

References:

- Spec: https://www.notion.so/defi-wonderland/Spec-2f09a4c092c781d89ba9f004f01a5cf5.

Optimism: Fixed by updating the Spec documentation.

Cantina Managed: Fix verified.

3.2.5 Event-name drift breaks spec-aligned indexers and monitoring

Severity: Informational

Context: PolicyEngineStaking.sol#L71-L79

Description: The spec/design flow expects `Linked/Unlinked` events for beneficiary lifecycle, but the contract uses `BeneficiarySet/BeneficiaryRemoved`:

```
// Spec/Design flows
MUST emit the `Linked` event ...
MUST emit the `Unlinked` event ...
```

```
event BeneficiarySet(address indexed staker, address indexed beneficiary);
event BeneficiaryRemoved(address indexed staker, address indexed previousBeneficiary);
//...
emit BeneficiaryRemoved(msg.sender, currentBeneficiary);
emit BeneficiarySet(msg.sender, _beneficiary);
```

Recommendation: Choose one event naming contract and keep it consistent across spec, design documents, interface docs and implementation. Either rename events in code to `Linked/Unlinked` or update documentation to the implemented `BeneficiarySet/BeneficiaryRemoved` naming.

Optimism: Fixed by updating the Spec documentation.

Cantina Managed: Fix verified.

3.2.6 Unnecessary `lastUpdate` reset in `_decreasePeData`

Severity: Informational

Context: PolicyEngineStaking.sol#L324-L329

Description: The `_decreasePeData` function unconditionally resets `lastUpdate` to `block.timestamp` when reducing an account's `effectiveStake`:

```
function _decreasePeData(address _account, uint128 _amount) internal {
    PEData storage pe = peData[_account];
    pe.effectiveStake -= _amount;
    pe.lastUpdate = uint128(block.timestamp);
    emit EffectiveStakeChanged(_account, pe.effectiveStake);
}
```

Since the `PolicyEngine` uses `lastUpdate` to determine the total weight of the staked position, resetting `lastUpdate` to the current timestamp effectively resets the staking weight.

This reset is unnecessary because a decrease guarantees the new `effectiveStake` is less than or equal to the previous value. The remaining stake has been continuously present since the previous `lastUpdate`, so the prior timestamp remains a valid (and more accurate) lower bound on how long the current stake has been held.

Recommendation: Preserve the existing `lastUpdate` on decreases, except when `effectiveStake` reaches zero (to avoid stale timestamps):

```
function _decreasePeData(address _account, uint128 _amount) internal {
    PEData storage pe = peData[_account];
    pe.effectiveStake -= _amount;
    if (pe.effectiveStake == 0) {
        pe.lastUpdate = uint128(block.timestamp);
    }
    emit EffectiveStakeChanged(_account, pe.effectiveStake);
}
```

Optimism: Fixed in commit 7630603 by preserving `lastUpdate` on non-zero decreases and only resetting it when `effectiveStake` reaches zero.

Cantina Managed: Fix verified.

3.2.7 Consider implementing a two-step ownership transfer pattern

Severity: Informational

Context: PolicyEngineStaking.sol#L169-L173

Description: The `transferOwnership` function transfers ownership in a single step. The current `owner` calls the function with a new address and ownership is immediately reassigned:

```
function transferOwnership(address _newOwner) external onlyOwner {
    if (_newOwner == address(0)) revert PolicyEngineStaking_ZeroAddress();
    emit OwnershipTransferred(_owner, _newOwner);
    _owner = _newOwner;
}
```

If the `owner` accidentally passes an incorrect address (e.g., a typo, wrong address from clipboard, or an address the caller does not control), ownership is irrevocably lost. There is no mechanism to recover from this, and all `owner`-gated functionality (`pause`, `unpause`, `transferOwnership`) becomes permanently inaccessible.

Recommendation: Implement a two-step ownership transfer pattern where the current `owner` nominates a `pendingOwner`, and the new `owner` must explicitly call a function to accept ownership to complete the transfer. This ensures the new `owner` controls the target address before ownership is finalized.

Optimism: Fixed in commit 564fc9a by replacing one-step ownership transfer with a two-step flow: `transferOwnership()` now sets a `pendingOwner`, and only that nominated address can finalize the transfer via `acceptOwnership()`.

Cantina Managed: Fix verified.

3.2.8 `PolicyEngineStaking` does not inherit `IPolicyEngineStaking`

Severity: Informational

Context: PolicyEngineStaking.sol#L15

Description: The `PolicyEngineStaking` contract inherits `ISemver` but does not inherit its own dedicated interface `IPolicyEngineStaking`. Without inheriting the interface, the compiler does not enforce that the contract correctly implements all functions defined in `IPolicyEngineStaking`. This means function signature mismatches (e.g., wrong parameter types, missing functions, incorrect return types) would go undetected at compile time, only surfacing at runtime or during integration.

Recommendation: Inherit `IPolicyEngineStaking` directly. Since `IPolicyEngineStaking` already extends `ISemver`, we can simply make the following change:

```
- contract PolicyEngineStaking is ISemver {
+ contract PolicyEngineStaking is IPolicyEngineStaking {
```

Optimism: Acknowledged.

Cantina Managed: Acknowledged.

3.2.9 Beneficiaries can reset a delegating `staker`'s `lastUpdate` by removing them from the allowlist

Severity: Informational

Context: PolicyEngineStaking.sol#L287-L297

Description: In `PolicyEngineStaking`, a `beneficiary` can reset the `lastUpdate` timestamp of any `staker` currently delegating to them by calling `setAllowedStaker(_staker, false)`. When a `staker` is removed from a `beneficiary`'s allowlist, the contract automatically reverts the `staker`'s beneficiary back to themselves, calling `_increasePeData` on the `staker`:

```
// PolicyEngineStaking.sol:setAllowedStaker()
if (!_allowed) {
   StakedData storage stakedData = stakingData[_staker];
   if (stakedData.beneficiary == msg.sender) {
     _decreasePeData(msg.sender, stakedData.stakedAmount);
     // ...
     stakedData.beneficiary = _staker;
     _increasePeData(_staker, stakedData.stakedAmount);
     // ...
   }
}
```

Since the `PolicyEngine` uses the `lastUpdate` of the `staker` to compute their stake weight, this allows the `beneficiary` to intentionally reset the `staker`'s stake weight. The contract protects against this form of griefing in the opposite direction via enforcing an `allowlist` to delegate to a beneficiary, but provides no protection for `staker`'s against their `beneficiary`.

This is an inherent trust assumption in the delegation model. A `staker` who delegates to a `beneficiary` implicitly trusts that the `beneficiary` will not remove them from the allowlist at a disadvantageous time.

Recommendation: Consider adding documentation so that stakers are aware of this risk before delegating.

Optimism: Fixed in commit 6f008ae by documenting in `setAllowedStaker()` that delegation carries a trust assumption: a beneficiary can remove a delegated staker at a disadvantageous time, which triggers `_increasePeData` on the staker and resets their `lastUpdate` / accumulated staking weight.

Cantina Managed: Fix verified.